The Machine-SUIF Static Single Assignment Library

Release version 2.02.07.15

Glenn Holloway
holloway@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

July 15, 2002

Abstract

The Machine SUIF static single-assignment library translates an optimization unit into or out of static single-assignment form (SSA), a constrained representation that supports many kinds of analysis and optimization. Conversion to SSA form produces an object of class `SsaCfg` that holds information about the transformed code, including both data-flow information for use during optimization and the maps needed to return the code to conventional form.

Our implementation follows the design described by Briggs *et al* [2] at Rice University. Their approach is flexible, allowing users to control the space/speed tradeoff. It also deals carefully with correctness and efficiency issues that arose in previous implementations.

This document describes how to use the SSA library, and it makes connections to the paper [2] and the implementation document [3] describing the Rice approach.

Contents

1 Introduction

Static single-assignment form scarely needs an introduction. In the relatively few years since its invention, it has become a popular representation for use in data-flow analysis and optimization. It is so robust and useful that the entire global optimization phase of the SGI Pro64 compiler is carried out in SSA form.

The Massively Scalar Compiler Project at Rice University, headed by Keith Cooper, has developed many optimizations based on SSA form. They have studied the issues in implementing the translation to and from SSA representation. Their findings are published in a journal article [2] and a detailed account, in "literate programming" style, of their implementation [3]. Both of these documents are available online.

Rice's design provides flexibility by allowing users to choose speed or space efficiency and to control whether to optimize during transformation and whether to retain data-flow information. It also addresses tricky correctness problems that arise when translating SSA code back to conventional, executable form. In general, that translation requires insertion of copy instructions to retain program semantics, and the Rice work shows how to minimize the number of copies used.

The Machine SUIF SSA library is based on the Rice design. In fact, the first draft of the code was written at Rice by a Todd Waterman, a student in Keith Cooper's group. For consistency with Machine SUIF conventions, we have reshaped the code and renamed some of the concepts, but we have retained the basic functionality and the efficient algorithms developed at Rice.

This document describes how to use the SSA library, but it is meant neither as an introduction to SSA-based optimization nor as a detailed rationale for the Rice approach. Any of several recent compiler textbooks can provide the former [1, 4, 5]. For the latter, read the Rice paper [2] and the Rice code description [3].[1] In the sections that follow, we point out connections between our implementation and theirs, and we also identify things that we've done differently.

Section 2 establishes a vocabulary. Section 3 is an introduction to the library's interface. Section 4 uses excerpts from a dead-code elimination pass to show how the interface is used. Section 5 is a reference guide to the interface. Section 7 gives a brief summary.

2 Terminology

We begin by establishing a vocabulary. Where possible, we use the same terminology used by Rice, which in turn reflects the SSA literatire for the most part. In some cases, however, we found conflicts with Machine SUIF conventions or customary usage too confusing, so we have made some changes.

We're careful in our use of the related terms *name*, *location*, and *value*. In non-SSA code, a name represents the contents of a storage location, but for any particular use of the name, we usually can't say which definition filled the location and thus gave the name its value. In SSA code, a name represents the value generated at a single definition point.[2]

We say that an "old name", one that occurs in non-SSA code, stands for a location, while a "new name", one in SSA code, stands for a value. In Machine SUIF, both kinds of names are represented as operands, i.e., they have type `Opnd`. The old names can be variable-symbol operands or register operands; these are the items SSA conversion pays attention to. The new names inserted by conversion to SSA form are always virtual registers, distinct from any that were in the original code.

In Rice's implementation, operands, and therefore names, in the sense above, are represented as integers. That's very handy for looking up operand attributes in tables. Our names are operands, but we provide a map from these names to small-integer "identifiers", or "ID"s.

A singular feature of SSA form is the *ϕ-function*, the fictitious statement that defines a new SSA value by multiplexing different values of the same original variable, each corresponding to a different incoming

[1] We need to mention both of Rice's documents frequently. For brevity, we'll refer to the published paper [2] as the "Rice paper" and to the implementation document [3] as the "Rice code".

[2] Rice uses the term "resource" instead of "location". Where we say 'value', they often say "definition" or "def".

control edge. The Rice paper, like most other literature, calls these "ϕ-functions", but the Rice code uses *ϕ-node*,[3] and we keep that term. We use the word *sources* to describe the inputs of a ϕ-node, since these "argument" values are analogous to the source operands of an instruction.[4] One or more of the sources of a ϕ-node may be *null*. That means that the underlying location has no definition along the incoming edge that corresponds to the source. A ϕ-node whose sources are all the same, ignoring nulls, is called a *useless* ϕ-node.[5] Useless ϕ-nodes may optionally be eliminated during conversion to SSA form.

Each use of a name is called an *occurrence* of that name. In SSA form, the sources of a ϕ-node are occurrences just like those in instructions. A definition point may be either an instruction or a ϕ-node. We use the term *operation* to cover both kinds of defining construct. An instance of class `Operation` is the handle of either an instruction or a ϕ-node.

The Rice paper describes three approaches to inserting ϕ-nodes, which vary in the time and space required to implement them. The *minimal* approach is so named because it requires less time, but it may insert lots of dead ϕ-nodes, i.e., one that define names that aren't used. The *pruned* style corrects that flaw by using liveness analysis to avoid inserting dead ϕ-nodes. It therefore saves space at compile time, but at the cost of doing data-flow analysis that might not otherwise be needed.

One of Rice's innovations is a third approach that they call *semi-pruned*. It's based on the fact that many of the names in a program being converted to SSA form are local to a single basic block. They are compiler-generated temporaries that are never live across a control-flow edge. Therefore, they never require ϕ-nodes. Semi-pruned form uses this observation to avoid inserting many of the ϕ-nodes that would be in minimal form without performing the liveness analysis needed for fully pruned form.

Figure 1.4 of the Rice code document illustrates the differences among the three flavors of ϕ-node insertion, and the text accompanying that figure talks about the settings in which each is most appropriate.

3 Overview of the Library's Interface

SSA form is a replacement intermediate representation. Class `SsaCfg` is a subclass of `AnyBody`, which means that one of its instances can serve as the body of an optimization unit, just like those of `InstrList` and `Cfg`.

SSA form construction starts from an existing CFG, which remains embedded in the `SsaCfg` object that results. You access it using the function `get_cfg`. Conversion to SSA form changes the code, however, so that the contents of the CFG no longer fully describe its semantics. The old names in the code are replaced by their new, SSA counterparts, but the ϕ-nodes that define some of the new names aren't inserted explicitly in the instruction stream. Instead, each basic block in the CFG may have an associated set of ϕ-nodes. The OPI includes functions that map from blocks to ϕ-nodes and vice versa. In general, optimizations that use SSA form need to be aware of the ϕ-nodes and they must use these OPI functions for dealing with them.

At the end of an SSA-based optimization pass, you can call the function `restore` to convert back to conventional, non-SSA form, so that the code in the CFG again expresses the full semantics of the unit being compiled. At that point, you would normally also store the CFG as the body of the unit (using `set_body`), and then discard the SSA object. Alternatively, you can leave the `SsaCfg` object as the body, for input to a later SSA-based pass.

Every SSA (new) name corresponds to an original (old) name, and when feasible, it's nice to restore the original names during conversion back from SSA form. That's the default behavior of the `restore` function. But an SSA-based optimization may make this impossible. Since mapping from SSA values to original names is many-to-one, there's no guarantee in general that every SSA-name occurrence falls at a

[3]Maybe "node" refers to an element of the internal representation of the program, or maybe it just means "list element", since a basic block may in general have a list of ϕ-functions. This latter use of "node" occurs repeatedly in the Rice code.

[4]The Rice code uses "parameter", abbreviated `parm`.

[5]The Rice code defines *useless* more aggressively, but notes that this may not be a good idea. We found cases in which the Rice definition caused to many ϕ-nodes to be eliminated.

point where the corresponding original location happens to hold the right value. (Figure 1.5 of the Rice code shows a simple example.) So `restore` takes a flag telling it not to restore the old names, but instead to leave the SSA names in the code. This means that ϕ-nodes have to be replaced by copy instructions inserted in predecessor blocks. The Rice group worked hard to make this efficient, and we retain their methods, but quite a few copy instructions may be inserted at this stage. Fortunately, many of these copies can usually be eliminated by later optimization passes.

SSA conversion can optionally develop and record use-def and def-use chains, although since each use-def "chain" has exactly one element, we don't call it a chain. The OPI includes function for accessing this data-flow information.

Other options at SSA construction time are to eliminate copy instructions and useless ϕ-nodes. A copy instruction becomes unnecessary if subsequent uses of its destination name are replaced by its source name. If this is done, the copy can be dropped. This is usually the right thing to do, but not always, because it means that `restore` can't restore old names. A *useless* ϕ-node is one whose source operands are identical. Folding of useless ϕ-nodes is always beneficial.

Sometimes, an optimization needs to introduce new names while the program is in SSA form and possibly to replace occurrences of existing names with the new ones. There are OPI functions that you call to record such changes in the SSA database. By convention, your optimization pass first modifies the code and related ϕ-nodes, and then calls methods to reflect the changes it has made. For example, when you replace one use occurrence by another, you call `record_use_swap` to update the internal records of your `SsaCfg` object. When you insert a completely new instruction or ϕ-node, you call `record_def` to record the new definition and `record_all_uses` to record the new use occurrences.

4 Excerpts from An Example Pass

An SSA-based optimization pass called `dcessa` is distributed with Machine SUIF. This is an implementation of the dead-code elimination algorithm given in Robert Morgan's textbook [4]. Without going into the details of the algorithm, we'll show how it makes use of the SSA library.

The conversion of an optimization unit's CFG to SSA form is carried out by these lines:

5 ⟨*Convert to semi-pruned SSA form* 5⟩≡

```
using namespace ssa;

unit_ssa = new_ssa_cfg(unit_cfg, cur_unit,
                       BUILD_SEMI_PRUNED_FORM | BUILD_DEF_USE_CHAINS |
                       PRINT_WARNINGS);
if_debug(5) {
    fprintf(stderr, "\nInitial SSA CFG:\n");
    fprint (stderr, unit_ssa);
}
```

The snippet above opens the `ssa` namespace so that SSA library tokens like `BUILD_PRUNED_FORM` can be used without qualification. Next it creates the `SsaCfg` object that will be used throughout the pass, and it immediately does the conversion by applying `build` to the unit's CFG. In this case, we opt to retain use-def and def-use information because it's needed for eliminating dead code. We also choose semi-pruned form to avoid the cost of doing liveness analysis. The extra ϕ-nodes that may result because we avoided full pruning aren't ultimately detrimental because at the end of this particular optimization, we can restore the original names instead of having to insert copy instructions to realize each ϕ-node. (We also choose *not* to fold away copy instructions during `build`, because that would inhibit the restoration of original names later.) Finally, when in high-verbosity debugging mode, we print the SSA representation on `stderr`.

Once the DCE pass has found an operation that must be preserved in the code being optimized, it goes through the input operands of that operation and marks the definition of each such operand as needing to be preserved itself. Here's the code that does this for one operand, called `opnd`:

6a ⟨*Mark necessary definition* 6a⟩≡
```
        Operation def = get_unique_def(ssa, opnd);
        if (!def.is_null())
            dce->mark_necessary(def);
```

Function `get_unique_def` takes the place of a use-def "chain"; i.e., it connects a use to its defining operation, which may either be a ϕ-node or an instruction. The snippet above first excludes operands (like immediate values) that have no definition, and then it deals separately with the two kinds of defining operations.

When applied to a ϕ-node, the DCE pass's `mark_necessary` function uses the name-to-identifier map stored in the `SsaCfg` object to obtain an integer ID uniquely associated with the name that the ϕ-node defines:

6b ⟨*Mark necessary* `operation` 6b⟩≡
```
    if (has_note(operation, k_necessary_operation))
        return;
    set_note(operation, k_necessary_operation, note_flag());
    worklist.push_back(operation);
```

In the above excerpt, the elements of `worklist` have class `Operation`, so that each repesents a handle on either a ϕ-node or and instruction.

Finally, here is the code that handles control dependences for a known-necessary `operation` that has just been popped from the worklist:

6c ⟨*Mark necessary CTIs* 6c⟩≡
```
        mark_controlling_ctis(get_parent_node(operation));
```

The helper `mark_controlling_ctis` that's used above takes a basic block and it makes sure that any control-transfer instruction (CTI) on which that block is control-dependent gets marked as necessary, i.e., not to be eliminated. The overloaded function `get_parent_node` maps either a ϕ-node or an instruction to its associated basic block.

Finally, at the end of the `dcessa` pass, we call `restore` to put the code back in conventional executable form:

6d ⟨*Restore conventional (non-SSA) form* 6d⟩≡
```
    unit_cfg = restore(unit_ssa);
    set_body(unit, unit_cfg);

    delete unit_ssa;
```

The `restore` function detaches the `Cfg` object from the `SsaCfg` object and returns the former. Otherwise, the subsequent `delete` statement would delete them both. We use `set_body` to make the CFG the body of the current optimization unit, and then we discard the SSA variant that it replaces.

In this example, we chose to revert to the original names of the program, which is safe with DCE because its only effect is to eliminate unneeded instructions, not to reorder the code or make substitutions. You can tell `restore` to retain the new names by passing it the Boolean argument `true`.

5 A Reference Guide to the Library's Interface

This section briefly describes the part of the OPI that deals with SSA representation. We give the declarations of relevant functions and types, followed by capsule summaries of their values and effects.

6e ⟨*Transformation to and from SSA form* 6e⟩≡

```
SsaCfg* new_ssa_cfg(Cfg*, OptUnit*, unsigned flags = 0);
Cfg* restore(SsaCfg*, bool not_old_names = false);
```

Function `new_ssa_cfg` transforms a CFG to SSA form and returns a `SsaCfg` object that embeds the modified CFG. It takes the current optimization unit as an argument to provide information about formal parameters. Acceptable values for its values flag argument are described just below.

Function `restore` transforms back to conventional form. It also disconnects and returns the embedded CFG, so that the SSA object can be reclaimed without affecting the CFG. By default, it restores the original names for SSA variables, but flag `not_old_names` causes it to keep the SSA names.

7a ⟨*Flags for SSA conversion* 7a⟩≡

```
namespace ssa {

    enum {
        BUILD_MINIMAL_FORM     = 1<<0,
        BUILD_SEMI_PRUNED_FORM = 1<<1,
        BUILD_PRUNED_FORM      = 1<<2,
        BUILD_DEF_USE_CHAINS   = 1<<3,
        FOLD_COPIES            = 1<<4,
        OMIT_USELESS_PHI_NODES = 1<<5,
        PRINT_WARNINGS         = 1<<6
    };

} // namespace ssa
```

The `flags` argument to `new_ssa_cfg` should be the bitwise union of one or more of the above-defined tokens. Note that these are in the `ssa` namespace. To use them, you can either qualify each with the prefix `ssa::`, or you can open the namespace with a `using namespace` declaration, as shown in Section 4.

- `BUILD_MINIMAL_FORM`, `BUILD_SEMI_PRUNED_FORM`, and `BUILD_PRUNED_FORM` are mutually exclusive. Minimal form ignores liveness in placing ϕ-nodes. Pruned form only inserts ϕ-nodes at merge points where the defined value is known to be live. Semi-pruned form is a compromise that doesn't require full-blown liveness analysis. It places ϕ-nodes only for names known to be live on entry to *some* basic block.
- `BUILD_DEF_USE_CHAINS` directs the library to record def-use mappings, which are available through the `get_use_chain` function (described below).
- `FOLD_COPIES` causes the transformation into SSA form to eliminate copy instructions involving SSA names by propagating the source of a copy to the occurrences of its destination name.
- `OMIT_USELESS_PHI_NODES` causes a ϕ-node to be omitted if every path to its basic block has the same SSA value for the original name.
- `PRINT_WARNINGS` turns on some warnings that are suppressed by default.

7b ⟨*Accessors for SSA form* 7b⟩≡
```
Cfg* get_cfg(SsaCfg*);

bool has_minimal_form(SsaCfg*);
bool has_semi_pruned_form(SsaCfg*);
bool has_pruned_form(SsaCfg*);

bool has_def_use_chains(SsaCfg*);
bool has_copies_folded(SsaCfg*);
bool has_useless_phi_nodes_folded(SsaCfg*);

int get_loc_count(SsaCfg*);                   // number of original locations
int get_value_count(SsaCfg*);                 // number of SSA defs

const OpndCatalog* get_value_catalog(SsaCfg*);
Opnd get_value_name(SsaCfg*, int value_id);     // map value_id to SSA name
CfgNode* get_def_block(SsaCfg*, int value_id);  // block holding value's definer

Operation get_unique_def(SsaCfg*, Opnd value_name);
const List<Operation>& get_def_use_chain(SsaCfg*, Opnd value_name);
```

Several functions simply report the inputs used to build SSA form. `get_cfg` returns the embedded CFG. The result of `has_minimal_form` reflects whether `BUILD_MINIMAL_FORM` form has been constructed. Similarly for `has_semi_pruned_form` and `has_pruned_form`. Likewise, `has_use_def_chains`, `has_copies_folded`, and `has_useless_phi_nodes_folded`, reflect the respective input flags.

The `get_loc_count` function returns the number of original names, while `get_value_count` gives the number of SSA definitions, i.e., the number of SSA values.

`get_value_catalog()` returns a map from SSA names to their integer identifiers. (The map is an `OpndCatalog`, which is defined in the `machine` library and is typically used in data flow analysis.) Function `get_value_name` inverts the map: it returns the SSA name for a given identifier. And `get_def_block` returns the CFG node with which an SSA definition (instruction or ϕ-node) is associated. (It returns `NULL` if the definition is that of a useless ϕ-node that has been eliminated.)

Finally, `get_unique_def` returns the defining operation for a name in SSA form, and `get_def_use_chain` returns the list of operations containing use occurrences of a name.

8a ⟨*Attachment of* `PhiNode`*s to blocks* 8a⟩≡
```
typedef List<PhiNode*>::iterator PhiHandle;

const List<PhiNode*>& get_phi_nodes(SsaCfg*, CfgNode*);
PhiHandle append_phi_node(SsaCfg*, CfgNode*, PhiNode*);
PhiNode* remove_phi_node(SsaCfg*, CfgNode*, PhiNode*);
```

The set of ϕ-nodes attached to a block is represented as a list, and the above functions allow you to access, extend and contract these lists

8b ⟨*Reflecting code changes* 8b⟩≡
```
void record_def(SsaCfg*, Opnd old_name, Operation, int dst_pos = 0);
void record_use_swap(SsaCfg*, Opnd out, Opnd in, Operation);
void record_all_uses(SsaCfg*, Operation);
```

The `record_` functions are used to inform an `SsaCfg` object about changes that you make in the SSA code.

Function `record_def` takes note of a new definition, in the form either of an instruction or a ϕ-node. It does not change the instruction or ϕ-node, but looks there to find the SSA name defined. You give it the `old_name` so that it can later restore original names. In the instruction case, you also provide the position of the relevant destination field in the instruction.

Function `record_use_swap` takes note of a replacement (`in` replacing `out`) among the use occurrences of SSA names in an instruction or ϕ-node. It doesn't alter the instruction or ϕ-node; it just updates internal records (e.g., def-use chains) to reflect the swap. If the name being swapped in is a newly-created SSA name, its definition must already have been processed by `record_def` before `record_use_swap` is called.

Function `record_all_uses` takes note of all the use occurrences within a new instruction or ϕ-node. It modifies internal records (e.g., def-use chains) to reflect insertion of the instruction or ϕ-node into the code.

9a ⟨*Printing the SSA CFG* 9a⟩≡

```
void fprint(FILE*, SsaCfg*);
```

`fprint` prints the CFG with ϕ-nodes for use in debugging.

9b ⟨*OPI functions for* `PhiNode`*s* 9b⟩≡

```
PhiNode* new_phi_node(int src_count);
int srcs_size(PhiNode*);
Opnd get_src(PhiNode*, int pos);
void set_src(PhiNode*, int pos, Opnd);
Opnd get_dst(PhiNode*);
void set_dst(PhiNode*, Opnd);
bool get_is_useless(PhiNode*);
void set_is_useless(PhiNode*, bool);
CfgNode* get_parent_node(PhiNode*);
void map_opnds(PhiNode*, OpndFilter&);
```

Function `new_phi_node` creates a ϕ-node with a given number of source names. Then you use other functions in the above list to fill in its fields. Most of these functions are analogous to their like-named counterparts for instructions.

9c ⟨*Class* `Operation` 9c⟩≡

```
class Operation
{
  public:
    Operation() :
        instr_handle(one_null_instr.begin()),
        phi_handle  (one_null_phi.begin())
      { }

    Operation(InstrHandle ih) { set_instr_handle(ih); }
    Operation(PhiHandle ph) { set_phi_handle (ph); }

    PhiHandle get_phi_handle() const { return phi_handle; }
    InstrHandle get_instr_handle() const { return instr_handle; }

    void set_instr_handle(InstrHandle ih)
        { instr_handle = ih; phi_handle = one_null_phi.begin(); }
    void set_phi_handle(PhiHandle ph)
        { phi_handle = ph; instr_handle = one_null_instr.begin(); }

    bool is_instr_handle() const
        { return *phi_handle == NULL && *instr_handle != NULL; }
```

```
    bool is_phi_handle() const
        { return *phi_handle != NULL && *instr_handle == NULL; }
    bool is_null() const
        { return *phi_handle == NULL && *instr_handle == NULL; }

    static list<Instr*>   one_null_instr;
    static list<PhiNode*> one_null_phi;

  protected:
    InstrHandle instr_handle;
    PhiHandle phi_handle;
};
```

Class `Operation` implements a "union" type each of whose instances is a handle on either a ϕ-node or an instruction.

10 ⟨*Functions on* `Operation`*s* 10⟩≡

```
Opnd get_dst(Operation, int pos = 0);
CfgNode* get_parent_node(Operation);
void map_opnds(Operation, OpndFilter&);

bool has_note (Operation, NoteKey);
Note get_note (Operation, NoteKey);
Note take_note(Operation, NoteKey);
void set_note (Operation, NoteKey, const Note&);
```

The helpers above extend a few overloaded functions to class `Operation`.

6 Adapter Passes

Machine SUIF includes simple passes for translating into and out of SSA form. `cfg2ssa` transforms a procedure already CFG form to SSA form. `ssa2cfg` goes the other way.

The `cfg2ssa` pass take a command-line flag corresponding to each of the flags accepted by `new_ssa_cfg`. For example, `-build_minimal_form` on the command line causes cfg2ssa to use the `BUILD_MINIMAL_FORM` flag when creating a `SsaCfg` object.

The `ssa2cfg` pass takes an optional `-restore_orig_names`. When present, it causes original location names to be restored during conversion from SSA form to conventional form.

7 Summary

The Machine SUIF SSA library provides efficient conversion to and from SSA form. It lets you control the trade-off between speed and space efficiency, and it gives you a number of options for performing optimizations during the conversion and for exposing data-flow information that it develops. The implementation closely follows the design laid out by Briggs *et al* [2, 3], which has proven effective in a number of optimization studies.

8 Acknowledgments

Our debt to the members of Keith Cooper's group at Rice University should be clear. Todd Waterman wrote the first version of the Machine SUIF implementation while an undergraduate member of that group. He used an alpha release of Machine SUIF, and the system was heavily revamped before its full release. So at Harvard, Kathleen Durant reshaped Todd's code to accommodate the changes in Machine SUIF and to conform with our more conventional (less "literate") style of organization and documentation.

This work has been supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.

References

[1] A. Appel. *Modern Compiler Implementation in {C,Java,ML}*. Cambridge University Press, 1998.

[2] P. Briggs, K. Cooper, T. Harvey, and L. Simpson. *Practical Improvements to the Construction and Destruction of Static Single Assignment Form.* Software Practice and Experience, 28(8), pp. 859-881, July 1998. Available via `http://www.cs.rice.edu/~harv/ssa.ps`.

[3] P. Briggs, T. Harvey, and L. Simpson. *Static Single Assignment Construction.* Implementation document, 1996. Available via `ftp://ftp.cs.rice.edu/public/compilers/ai/SSA.ps`.

[4] Robert Morgan. *Building an Optimizing Compiler.* Digital Press, 1998.

[5] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.